REMARKS

Claims 1-28 remain pending in the Application. Claims 1-28 stand rejected by the examiner. Applicant traverses the rejections of claims 1-28.

Claim Rejections

Claims 1-7, 9-24 and 26-28 stand rejected as being anticipated by "Dynamic Class Loading in the Java™ Virtual Machine" by Liang et al. (hereinafter referred to as Liang). Claims 8 and 25 stand rejected under 35 U.S.C. § 103(a) as being unpatentable over Liang in view of U.S. Patent 6,675,381 to Yamaguchi. These rejections are traversed.

Claims 1 and 16 provide approaches for loading an object class into computer memory for access by applications. A destructible class loader is created such that the destructible class loader is accessed instead of another class loader when a class loading operation is utilized. The pre-existing class loading hierarchy is modified so that the destructible class loader is utilized within the hierarchy.

For example, the destructible classloader can be made to "look like" a system classloader, such as the system classloader in Java™, and can be released and another new classloader may be added with class changes at anytime (without having to stop and restart the application to install the class changes). The releasing and recreation are performed transparently in that pre-existing classloaders do not need to know that the new "destructible" classloader has been inserted into the class loading hierarchy. The new "destructible" classloader also is not required to know the entire structure of the pre-

existing classloading hierarchy since it (and not the system classloader) is automatically returned via the system classloader operation.

Figure 4 of applicant's specification illustrates an example of a modification of "a pre-existing class loading hierarchy that specifies parent-child relationships of class loaders" (as recited in claim 16). As illustrated in applicant's figure 4, arrow (80) indicates that classload requests in the modified classloading hierarchy travel from a requesting classloader down to the bootstrap classloader (50) before a classload is attempted. Arrow (82) indicates that failed classload requests travel back to the original classloader for additional searches. For example, if an object within the extensions classloader (48) makes a classload request, the object can ask the extensions classloader (48) to process the load request. The extensions classloader (48) asks the bootstrap classloader (50) first to load the class. If the bootstrap classloader (50) is unsuccessful, then the extensions classloader (48) will attempt to load the class. If that fails, the classload is rejected. Instead, the object can ask the system classloader (46) to load the class. Instead of the system classloader (46) being returned as the classloader (as would be in prior systems' approaches), the destructible classloader (34) is returned due to the swapping/switching performed by the approach of claim 16.

The examiner maintains that the Liang reference teaches the limitations of claims 1 and 16. For example, the examiner maintains that the modification of a pre-existing class loading hierarchy that specifies parent-child relationships of class loaders (as recited in claim 16) is disclosed in the Liang reference by allowing a user to define their own classloaders, and is further shown in Section 2.3, page 38, wherein "Subclasses of ClassLoader can override the definition of loadClass, thus providing a user-defined

loading policy." Applicant respectfully disagrees with this position. The Liang reference may discuss a user defining a load policy, but it does not address modifying the pre-existing class loading hierarchy, let alone the specific approaches recited in applicant's claims 1 and 16. As an illustration, the Liang reference uses the pre-existing class loading hierarchy shown in Figure 2 on page 37 of the reference. The Liang reference does not discuss modifying this pre-existing class hierarchy as required by claims 1 and 16. Section 2.3 cited by the examiner further supports this. In Section 2.3, the Liang reference provides a code example that shows that the Liang reference is not adding a new class loader to the pre-existing class loading hierarchy. Instead, the Liang reference is extending an already existing classloader: "class MyClassLoader extends ClassLoader..." (see page 38 of the Liang reference). Simply extending a classloader does not address how it is used in the hierarchy. The classloader used in the example has no hierarchy. Accordingly, the hierarchy shown in Figure 2 of the Liang reference is not changed (e.g., a new class loader is not inserted as illustrated in Figure 4 of applicant's specification). Because of such significant differences between the Liang reference (whether considered alone or in combination with the other cited references) and claims 1 and 16, claims 1 and 16 are allowable as well as their respective dependent claims.

[CONTINUED ON THE NEXT PAGE]

CONCLUSION

For the foregoing reasons and others, the Liang reference (whether considered alone or in combination with the other cited references) does not disclose, teach, suggest such limitations. Accordingly, Applicant respectfully submit that claims 1-28 are allowable, and the Examiner is respectfully requested to pass this case to issue.

Respectfully submitted,

Date: 1/13/05

By: _____

John V. Biernacki
Reg. No. 40,511
JONES DAY
North Point
901 Lakeside Avenue
Cleveland, Ohio 44114
(216) 586-3939